Applications of Neural Networks to Solve the Differential Equations of Magnetohydrodynamics

Ph.D. Thesis Proposal (DRAFT)

Eric Winter (George Mason University)

March 22, 2020

Abstract

The proposed work will provide a novel capability for efficiently solving the equations of magnetohydrodynamics (MHD) for space plasma scenarios. Previous work has shown the efficacy of using neural networks with one or more hidden layers to solve simple ordinary and partial differential equations, but these techniques will be extended to create the first neural network-based solution of the coupled set of 2nd-order partial differential equations which define MHD in space. Several distinct MHD regimes will be examined. The technique will be applied first to relatively simple model scenarios, such as the steady-state solar wind flow, and the propagation of magnetic flux lines from the solar photosphere to the corona. Then the mechanism will be extended to more complex scenarios with dynamic boundary conditions, such as coronal mass ejections, and the onset and propagation of plasma wave phenomena. These techniques will provide a set of solutions that are completely mesh-free, and differentiable. In many situations, we expect the neural network approach will prove to be significantly faster than the currently-standard techniques of the finite difference method (FDM) and the finite element method (FEM). The neural network method will be shown to be more efficient in its use of computational resources, and provide a solution which is at least as accurate as solutions from traditional methods. This work will provide a critical, independent solution of the MHD equations, which can be used in conjunction with existing heritage models to improve the speed and accuracy of analyses of space plasma and space weather events.

Contents

1 Overview

1.1 Scientific Rationale

The differential equations of magnetohydrodynamics (MHD) govern the behavior of all space plasmas, and thus are the primary equations which govern the behavior of space weather phenomena. As our reliance on space-based assets for communication, observation, and exploration has increased, the vulnerability of those assets to severe space weather events has also increased [49]. If the speed and accuracy of models of space weather events can be improved, the risk to personnel and material assets in space can be reduced by providing increased warning time, allowing improved preparations to withstand the potential consequences of the event.

The MHD equations combine the mechanics of fluid flow with Maxwell's equations of electrodynamics. Similar to the Navier-Stokes equations, the MHD equations describe the effects of the conservation of mass, momentum, and energy. But the MHD equations also include terms for the conservation of electric charge and magnetic flux, and the interdependence of currents, electric, and magnetic fields. The detailed form of the equations is dependent on the assumptions made in their derivation. For example, for flow of an adiabatic, shock-free plasma, with a single ion species, the MHD equations [17] become:

Gauss' Law

$$\epsilon_0 \nabla \cdot \mathbf{E} = n_i q_i + n_e q_e \tag{1}$$

Faraday's Law

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{2}$$

Magnetic divergence (no magnetic monopoles)

$$\nabla \cdot \mathbf{B} = 0 \tag{3}$$

Ampère's Law

$$\frac{1}{\mu_0} \nabla \times \mathbf{B} = n_i q_i \mathbf{v}_i + n_e q_e \mathbf{v}_e + \epsilon_0 \frac{\partial \mathbf{E}}{\partial t} \tag{4}$$

Conservation of mass ($j = i|e$)

$$\frac{\partial n_j}{\partial t} + \nabla \cdot (n_j \mathbf{v}_j) = 0 \tag{5}$$

Conservation of momentum ($j = i|e$)

$$m_j n_j \left[\frac{\partial \mathbf{v}_j}{\partial t} + (\mathbf{v}_j \cdot \nabla) \mathbf{v}_j \right] = q_j n_j (\mathbf{E} + \mathbf{v}_j \times \mathbf{B}) - \nabla p_j \tag{6}$$

Equation of state ($j = i|e$)

$$p_j = C_j n_j^{\gamma_j} \tag{7}$$

Taken as a set, these 18 scalar equations govern 16 scalar unknowns (considering vector components as scalars). However, (1) and (3) are superfluous, since they can be obtained from the divergence of (4) and (2), respectively. We are thus left with a set of 16 equations and the same number of unknowns, so a solution is (theoretically) possible. This solution incorporates the effects of the electric and magnetic fields on plasma motion, and the effects of plasma motion on the electric and magnetic fields, in a self-consistent manner.

These equations can be used as the starting point for the investigation of specific plasma environments and phenomena. For example, space plasmas can usually be treated as collisionless, and (6) becomes:

$$\mathbf{E} + \mathbf{v}_j \times \mathbf{B} = 0 \tag{8}$$

This result leads to the "frozen-in" condition, in which the magnetic flux is essentially carried along by the frozen plasma. The magnetic field of the solar plasma is one if the primary inputs for models of space weather phenomena in the vicinity of planetary magnetospheres. For example, when the solar and terrestrial fields are roughly anti-parallel, magnetic field reconnection events can become more frequent and more powerful, leading to solar storms and substorms.

The MHD equations can be used to model regions progressively more distant from the Sun. For example, a simpliified two-dimensional MHD model can be used to examine the evolution of active region magnetic fields in the solar photosphere [46]. In this case, the analysis occurs on a two-dimensional surface in spherical coordinates (9).

$$\begin{aligned} \frac{\partial B_r}{\partial t} + \frac{1}{R_S \sin\theta} \frac{\partial}{\partial \theta} \left(v_\phi B_r \sin\theta \right) + v_\theta \frac{\partial B_r}{\partial \phi} \\ + \frac{\eta}{R_S^2 \sin\theta} \frac{\partial}{\partial \theta} \left(\sin\theta \frac{\partial B_r}{\partial \theta} \right) + \frac{\eta}{R_S^2 \sin^2\theta} \frac{\partial^2 B_r}{\partial \phi^2} = S(\theta, \phi, t) \end{aligned} \tag{9}$$

where η is the diffusivity, and S is a source term representing the creation of new active regions. This equation can be used to model the evolution of the photospheric magnetic field represented by solar synoptic maps.

Moving further outward, the connection between the surface field and the coronal field can be modeled with the potential field source surface (PFSS) model [4]. The PFSS model is a conceptually simple model used to estimate the solar coronal magnetic field from observations of the surface magnetic field. In this model, the static coronal field is derived from a scalar magnetic potential field:

$$\mathbf{B} = -\nabla\phi \tag{10}$$

$$\nabla^2\phi = 0 \tag{11}$$

$$B_r = -\frac{\partial\phi}{\partial r} \tag{12}$$

$$B_\theta = -\frac{1}{r}\frac{\partial\phi}{\partial\theta} \tag{13}$$

$$B_\phi = -\frac{1}{r\sin\theta}\frac{\partial\phi}{\partial\phi} \tag{14}$$

Despite its greatly simplified structure, the PFSS has proven so useful that it is included in the catalog of standard models offered by the Community Coordinated Modeling Center (see discussion of the CCMC below).

A more rigorous model of the coronal magnetic field, developed by [21], provides additional levels of detail in the model, including predictions of regions of trapped plasma, and the high-latitude "last closed field lines":

Conservation of mass

$$\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\mathbf{u}) = 0 \tag{15}$$

Conservation of momentum

$$\frac{\partial(\rho\mathbf{u})}{\partial t} + \nabla\cdot(\rho u\mathbf{u}) = -\nabla P - \frac{GM_S}{r^2}\rho\hat{\mathbf{r}} + \mathbf{j}\times\mathbf{B} \tag{16}$$

Ampère's Law

$$\mathbf{j} = \frac{1}{\mu_0}\nabla\times\mathbf{B} \tag{17}$$

Magnetic induction

$$\frac{\partial B}{\partial t} = \nabla\times(\mathbf{u}\times\mathbf{B}) \tag{18}$$

Ideal gas law

$$P = \frac{nkT}{m} \tag{19}$$

Beyond the corona, the solar atmosphere merges into the solar wind. The solar wind is typically modeled as a high-speed collisionless plasma with a frozen-in magnetic field. Various forms of the MHD equations are implemented in a variety of models used to analyze space plasmas. Models used for day-to-day space weather monitoring and prediction typically have a longer heritage than research models. For example, the NOAA Space Weather Prediction Center [14] maintains a stable of operational models and a large archive of historical data of solar activity and space weather. These products are used to produce short-term (3-day) and long-term (27-day) forecasts of solar activity and space weather conditions, which are also used as the basis for geomagnetic storm and other space weather warnings.

An example of these models is the WSA-ENLIL combined model [15]. This model has been used operationally at the NOAA SWPC since 2011. It is a semi-empirical model that predicts the speed, density, temperature, magnetic field, and other parameters of the "ambient" (non-CME) solar wind at 1 AU, and is thus a useful predictive tool for terrestrial space weather events. This model was deployed because full three-dimensional MHD simulations still take too much time to set up, run, and validate. The modeling system consists of two main parts: (1) a semi-empirical near-Sun module (WSA - Wang-Sheeley-Arge [5],[28]) that approximates the outflow at the base of the solar wind; and (2) ENLIL [42], a sophisticated three-dimensional magnetohydrodynamic numerical model that simulates the resulting flow evolution out to 1 AU. In addition to the SWPC, a wide variety of modern and legacy research models are available at the NASA Community Coordinated Modeling Center [13] at the NASA Goddard Space Flight Center. The results of these models are made available with scheduled and on-demand runs for the community, as well as an extensive archive.

1.2 Solving Differential Equations

The MHD equations (in any form) constitute a set of coupled partial differential equations (PDEs) that are second-order in spatial variables and first-order in time. Additionally, since these equations arise from conservation laws, the partial differential equations themselves are *hyperbolic*. Hyperbolic equations result in solutions with propagating behavior - a wave equation is a typical case. Solutions may be found using any of the current standard techniques for PDEs. Chief among these are the finite difference method (FDM) and the finite element method (FEM).

In the finite difference method [24], derivatives of the function field are approximated by difference equations using Taylor series. The difference equations are defined at a mesh of grid points in the problem domain. The result is a system of linear algebraic equations that must be solved to generate the solution at the selected grid points. Solution at non-grid points must be estimated using an interpolation scheme, or computing a new solution with an new grid. Finite difference methods have the advantage of conceptual simplicity and well-developed computational techniques. However, finite difference methods can be inefficient in situations where boundary conditions or physical parameters are dynamic, since any change requires a new solution of the entire set of coupled difference equations. Additionally, parallelization of the FDM can be a complex endeavor, since each mesh element depends on its neighboring elements. To avoid saturation of the parallel computing system, the domain mesh must be carefully partitioned among the available computation threads to minimize inter-thread communication during the solution process, e.g. [47].

In the finite element method [20], the problem domain is also subdivided into a mesh, and the differential equations of the system are defined for each subvolume of the domain. The collection of equations is then used to define a global error function. This error function is then minimized by an appropriate

method, and the parameter vector which minimizes the error function defines the solution to the problem. Like FDM, changes in location, system parameters, or boundary conditions require an interpolation scheme or a completely new solution cycle. FEM solutions also suffer from the same parallelization difficulties as FDM, leading to complicated code for large problems. Both FDM and FEM techniques suffer from increased approximation error as a function of time [12], which can lead to additional recomputations of the solutions for new boundary conditions. Adaptive mesh refinement, e.g. [9], can improve the quality of the solution in regions of rapid change, but solutions are still only available at the selected grid points. Several packages are available to address these issues, such as the widely-used PETSc toolkit [51], which provides support for the parallel distribution of matrices used in finite element models.

Since the late 1980s, new approaches for the solution of differential equations have been developed which take advantage of neural networks. There are a variety of techniques available, but all take advantage of one of the most mathematically useful aspects of neural networks - the ability to act as a "universal approximator" [25]. This capability was later shown to extend to approximation of the derivatives of the function [26]. The development of this field has been surveyed in recent comprehensive reviews [56, 16]. Neural network techniques generate mesh-free solutions which are differentiable to a useful extent, and are easily parallelized on modern multicore computers [48]. Drawbacks can include lengthy training times, especially in cases where local minima in the error function are encountered. Such situations can exacerbate the already long training times which result from solving higher-dimensional problems - the so-called "curse of dimensionality". However, neural network techniques have shown promise in improving performance over traditional finite difference solutions of PDEs of high dimensionality (Figure 1).

When the boundary conditions are dynamic, there are two basic modifications which can be made to the neural network solution. First, solutions can be generated at a representative set of times, and the solutions between those times interpolated as needed. Second, the dynamic boundary conditions themselves can be incorporated into the neural network solution itself, using either a functional or a data-based representation [31].

1.3 Objectives and Impact

1.3.1 Expand the Neural Network Computational Approach to the Coupled PDEs of MHD

The nature of a neural network solution provides two important benefits to solving differential equations. First, since the solution is mesh-free, no interpolation or recomputation is required to generate solution values at points not used in the training process. Second, since the neural network approach provides an analytical form for the solution, the solution is easily differentiable. Previous work has claimed that the solutions are "infinitely" differentiable [31], and this is mathematically correct. However, experience gained during the early phases

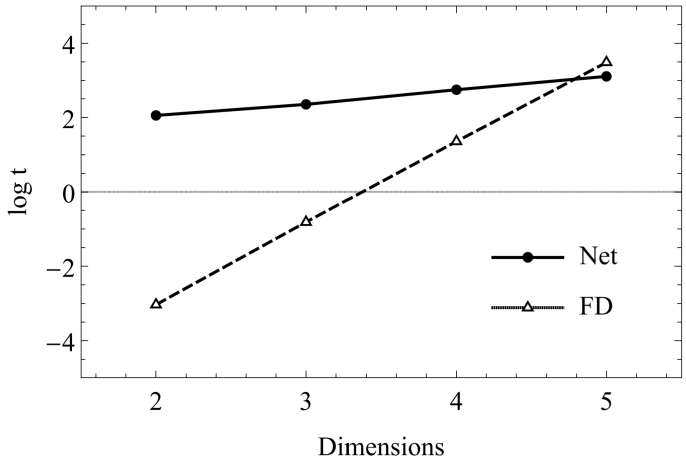

Figure 1: A comparison of solution times for neural network and 2nd-order finite difference methods for increasing problem dimensionality. The data were generated by specifying the analytical form of scalar functions of up to 5 independent variables, and substituting these solutions into linear and nonlinear forms of the Poisson equation. Training and testing points were randomly distributed on the surface of an n-dimensional hypersphere. Neural networks with up to 6 hidden layers were then trained to generate the solutions to compare against finite-difference solutions. Networks were trained using a GPGPU-equipped computer, and 32-bit precision. [6]

of this work have shown that the accurate differentiability extends only over a small range of orders. In the problems examined to date (typically simple second-order PDEs), the numerical values of the computed derivatives become less accurate as the order of differentiation is increased. One of the goals of this work is to quantify the accuracy of the derivatives of the neural network solution. This investigation will provide constraints on the degree of differentiation that can be usefully applied to a neural network solution. These constraints will improve the ability to match any Neumann boundary conditions set for the problem, which will be particularly useful when estimating mass and energy fluxes at the boundaries of the problem domain.

Every increase in model computation speed will provide increased lead time to prepare for a space weather event. Neural network methods are inherently logically parallel - computations at each node are completely independent of any other node in the same layer. Neural networks are thus more easily adapted to take advantage of parallel computing hardware. This will allow rapid scaling of problem size and solution speed, as the neural network software can take advantage of both local (multi-core, multi-thread, and GPU/accelerator) and non-local (multi-node) parallelism. This capability will be particularly important for three-dimensional MHD problems, for which a large number of training points will be required.

1.3.2 Demonstrate the Improved Utility and Efficiency of Compact Neural Network Solutions

Current MHD models used for space plasma analysis utilize well-tested methods from FDM and FEM. However, the amount of time needed to generate the model results, as well as the storage requirements for the resulting large datasets, can limit the amount of time available to analyze the results within a useful time frame, such as an impending severe space weather event. The parameter vectors generated by the neural networks will encapsulate a compact set of solutions for the MHD equations under physically realistic conditions. These compact solutions will allow focused investigations into the specific spatial and temporal regions that are of greatest interest to the researcher.

The parameter vectors will be generated over a range of parameter spaces, from the solar photosphere to the terrestrial magnetosphere interface. Existing standard methods, such as those discussed in the previous section, will be used as a basis for comparison for the neural network solutions. At each stage, the neural network results will be compared to equivalent results using FDM and/or FEM, to discover the parameter spaces where each method is the faster solution. The result will be a set of neural network alternative solutions for each of the space plasma regimes examined.

The software developed so far for this effort [55] is open-source, and made available for community use on GitHub [22]. Over the past two years, several experimental versions of the software for this effort have been developed. These early versions of the code have already been downloaded and/or forked several times from GitHub and used by other researchers. The rapidly-moving field

of machine learning requires this approach to ensure that the latest theoretical advances are rapidly incorporated into the software.

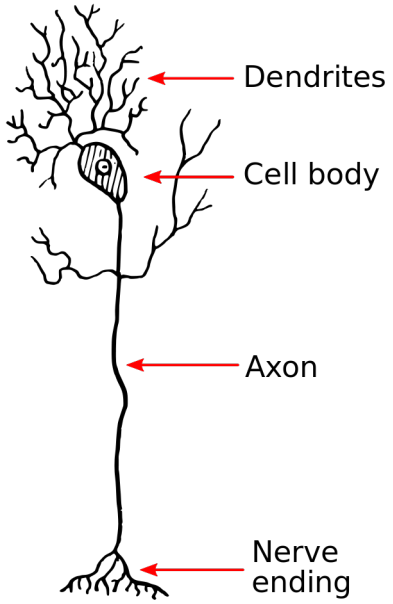

Figure 2: The components of a biological neural network [53].

2 Background and Previous Work

2.1 Neural Networks

2.1.1 General Introduction

The concept of computational neural networks was first introduced in the 1940s [35]. Biological neural networks are composed of a large number of neurons, connected by axons and dendrites (Figure 2). Dendrites bring signals to the neuron, where they are processed as a group to generate a single output, which is then transmitted to one or more subsequent neurons by the (often branching) axon.

A single computational neural network node is analogous to a biological neuron, in that it receives a fixed number of inputs, and computes a weighted combination z (the *activation*) of the inputs x (Figure 3). The node then applies a transfer function $\sigma(z)$ to the activation to compute a single output. This output is then transmitted to one or more subsequent nodes for further processing. The simplest version of this process can be represented as a weighted sum z_k of inputs x_j, followed by a binary comparison to a threshold value θ_k (20).

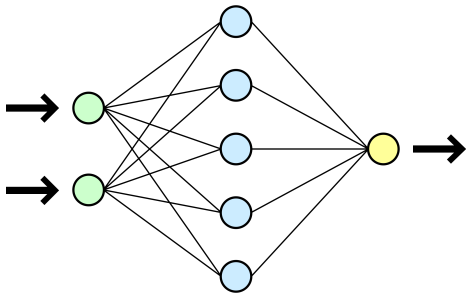

Figure 3: The components of a computational neural network [54].

$$z_k = \sum_{j=1}^{m} w_{jk} x_j$$
$$\sigma(z_k) = (z_k \geq \theta_k) \ ? \ 1 : 0 \tag{20}$$

where j is an index over the m inputs, and k is an index over the H network nodes. In this simple model, if the input exceeds a threshold value θ_k, the node "fires", sending its output (unity) to subsequent nodes.

While conceptually simple, and useful in discrete classification problems, this approach has limited usefulness when the output values of the network form a continuum. With a continuous instead of a binary transfer function applied in a node, a wider variety of behaviors can be simulated. The simplest continuous transfer function is simply the identity function - the output of the node is the weighted sum of the inputs (Equation 21).

$$\sigma(z_k) = z_k \tag{21}$$

However, this technique can lead to numerical instability due to its unbounded nature. A better approach is to apply a bounded, smoothly-varying, monotonic transfer function, such as the sigmoid function (22).

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{22}$$

This function enforces a limiting behavior - the output forms a bounded continuous distribution between 0 (for $z \to -\infty$)), and 1 (for $z \to \infty$)). Variations on this theme are possible. For example, the sigmoid limits can be $[-1, 1]$, or can be more complicated, e.g. using $tanh()$.

As computer hardware advanced, the expectations of the accomplishments of neural networks increased. Failure to rapidly achieve those expectations led to a relatively moribund period for the field [40]. In the 1980s, interest in the capabilities of neural networks was rekindled, in part, by new theoretical developments, and the precipitous decline in the cost of computer hardware [3].

2.1.2 Neural Networks as Universal Approximators

The renaissance of interest in neural networks included expansion of the domain of problems to which they could be applied. [25] showed that even a simple neural network architecture - a feedforward network with a single hidden layer - could act as a universal approximator for smooth scalar functions. In effect, the network could be thought of as an expansion of a function using a set of arbitrary basis functions. Subsequent theoretical work showed that simple neural networks could be used for a wide variety of function approximation problems, e.g. [10], [11], [52], and others.

2.1.3 Using Neural Networks to Solve Differential Equations

The universal approximation capability of neural networks was eventually turned to the solution of differential equations. A wide variety of neural networks have been developed to address these problems. [37] published some of the earliest results for nonlinear ordinary differential equations, investigating the utility of different piecewise-linear activation functions. [23] used multilayer neural networks to solve the differential equations involved in a nonlinear control system. [27] and [32] applied radial basis function (RBF) neural networks to the numerical solution of elliptic PDEs, and [29] provided an extensive review of RBF methods for PDEs. [50] applied neural networks to the solution of the governing differential equations of nonlinear (chaotic) systems. [41] used the constrained backpropagation (CPROP) technique with partitioned short- and long-term memory nodes to solve PDEs with dynamic boundary conditions. [59] investigated the use of BAM (Bidirectional Associative Memory) networks for the solution of nonlinear differential equations in control problems. [2] used a finite-difference method built on a Hopfield neural network to solve a variety of PDEs. [43] and [45] further developed the CPROP technique to solve nonlinear elliptic and parabolic PDEs. [34] used the functional expansion block in a Chebyshev neural network to expand the scalar input into a set of Chebyshev basis functions to solve singular Emden-Fowler differential equations. [44] used a constrained integration method to transform a partial differential equation to a system of ordinary differential equations which could then be integrated to find the neural network weights. [19] used a cellular neural network to solve hyperbolic PDEs arising in a distributed control system. [38] developed a useful approach for 2nd-order PDEs using least-squares support vector machines (LS-SVM), which allows for a solution by solving a linear system of equations. [1] used a neural network with several different training methods to find solutions for a fifth-order boundary value problem arising in the modeling of induction motors.

The conceptually simplest method of solving differential equations using neural networks is the "trial function" method, first proposed by [39], and generalized by [31]. In this method, an approximate solution, incorporating the neural network output, is substituted into the differential equation in place of the true solution. Just as for any other model-fitting procedure, the network

parameters are then adjusted, i.e. the network is trained, until the trial solution is sufficiently close to the actual solution. An early application of this method was solving for the eigenvalues of the Schrodinger equation [30].

Many variations on this technique have been developed. [33] showed the utility of the trial function approach for sets of ODEs of up to 4th order. [58], [57] used the trial-function method along with a kernel least mean square algorithm to solve 1st- and 2nd-order ODEs. [18] illustrated the use of the trial function method on a 2-D form of Laplace's equation. [7] used a trial-function method with a network with two hidden layers to generate solutions of the 2-dimensional Navier-Stokes equations for electroosmotic flow in a microchannel. [6] used a deep-network approach with trial functions to generate solutions for PDEs of up to 5 dimensions. [8] further developed the technique to allow it to be used over arbitrary spatial boundaries. A different technique for non-orthonormal problem domains was developed by [36].

2.2 The Trial Function Method

A generalized differential equation G of a scalar function $\psi(\mathbf{x})$ can be written in the form:

$$G(\mathbf{x}, \psi, \nabla\psi, \nabla\nabla\psi, ...) = 0 \tag{23}$$

In the trial function method, an approximate trial solution ψ_t is substituted for the desired solution ψ in the differential equation:

$$G(\mathbf{x}, \psi_t, \nabla\psi_t, \nabla\nabla\psi_t, ...) = 0 \tag{24}$$

Since the analytical form of the trial function is known, the parameters of the trial function can be adjusted until the differential equation is satisfied to the desired level of precision, while also satisfying any boundary conditions.

One particularly simple and useful form for the trial function is [31]:

$$\psi_t(\mathbf{x}, \mathbf{p}) = A(\mathbf{x}) + P(\mathbf{x})N(\mathbf{x}, \mathbf{p}) \tag{25}$$

where the function $A(\mathbf{x})$ incorporates all of the boundary conditions, and $N(\mathbf{x}, \mathbf{p})$ is the scalar output of a neural network with inputs $\mathbf{x}$ and parameters $\mathbf{p}$. The boundary condition function, $A(\mathbf{x})$, and the network coefficient function, $P(\mathbf{x})$, are functions only of the independent variables $\mathbf{x}$. The trial solution $\psi_t(\mathbf{x}, \mathbf{p})$ must satisfy the boundary conditions. Because the output $N(\mathbf{x}, \mathbf{p})$ of the neural network is unconstrained, the functions $A(\mathbf{x})$ and $P(\mathbf{x})$ must meet the following conditions to satisfy the differential equation G:

1. $P(\mathbf{x})$ must vanish at the boundaries.
2. $A(\mathbf{x})$ must satisfy the boundary conditions.

Both functions can have an arbitrary finite value within the domain.

Consider an arbitrary 2nd-order partial differential equation. In the following development, we assume the problem domain has been scaled to an orthogonal normalized domain $x_j \in [0,1] \; \forall \; 1 \leq j \leq m$, where m is the number of independent variables in the differential equation.

Once the form of the trial function has been determined, it can be used to create an error function for the network:

$$E = \sum_{i=1}^{n} G(\mathbf{x}_i, \psi_{ti}, \nabla\psi_{ti}, \nabla\nabla\psi_{ti})^2 = \sum_{i=1}^{n} G_i^2 \tag{26}$$

where n is the number of training points used by the neural network. This error function is then used as the metric, or cost function, for training the network. This is merely the simplest possible form for the error function - many variations on this approach are possible. For example, error functions can incorporate weighted contributions from the approximated values of $\nabla\psi_{ti}$, $\nabla\nabla\psi_{ti}$, and other higher-level derivatives.

Training of the network can take several forms. The simplest approach is the so-called "delta rule", in which the parameters are adjusted by a small amount after each pass through the training loop:

$$p_{new} = p + \eta \frac{\partial E}{\partial p} \tag{27}$$

where p represents any parameter of the neural network, and η is the "learning rate". To ensure stability, the learning rate is usually set to a small value (usually $\eta < 1$). The error function derivatives are computed using the form of the differential equation:

$$\frac{\partial E}{\partial p} = 2 \sum_{i=1}^{n} G_i \frac{\partial G_i}{\partial p} \tag{28}$$

Since the explicit form of G and the trial function ψ_t are known, and the structure of the neural network is known, all required derivatives of the error function are available in analytical form. However, they can also be estimated numerically when required for efficiency constraints.

The delta method has the advantage of conceptual and computational simplicity. However, simplicity comes at the cost of long training times. A better approach is to use any of a number of established numerical optimization techniques to train the network. In such cases, the Jacobian of the error function is constructed and used in the optimization of the network parameter values. Note that the Jacobian is a vector when the solution is a scalar field, and a matrix when the solution is a vector field.

To develop the Jacobian, we begin by applying the chain rule to the trial function in the error derivative (assume a 2nd-order PDE with no cross-partials):

$$\begin{aligned}\frac{\partial G_i}{\partial p} =& \sum_{j=1}^{m} \frac{\partial G_i}{\partial x_{ij}} \frac{\partial x_{ij}}{\partial p} + \frac{\partial G_i}{\partial \psi_{ti}} \frac{\partial \psi_{ti}}{\partial p} + \\ & \sum_{j=1}^{m} \frac{\partial G_i}{\partial \nabla_{ij} \psi_{ti}} \frac{\partial \nabla_{ij} \psi_{ti}}{\partial p} + \sum_{j=1}^{m} \frac{\partial G_i}{\partial \nabla_{ij}^2 \psi_{ti}} \frac{\partial \nabla_{ij}^2 \psi_{ti}}{\partial p}\end{aligned} \tag{29}$$

Since the input variables x_{ij} are independent of the network parameters p, this equation becomes:

$$\frac{\partial G_i}{\partial p} = \frac{\partial G_i}{\partial \psi_{ti}} \frac{\partial \psi_{ti}}{\partial p} + \sum_{j=1}^{m} \frac{\partial G_i}{\partial \nabla_{ij} \psi_{ti}} \frac{\partial \nabla_{ij} \psi_{ti}}{\partial p} + \sum_{j=1}^{m} \frac{\partial G_i}{\partial \nabla_{ij}^2 \psi_{ti}} \frac{\partial \nabla_{ij}^2 \psi_{ti}}{\partial p} \tag{30}$$

The analytical forms for the partial derivatives of G_i are found using the form of the original differential equation, and are specific to the equation being solved. The individual terms of ψ_{ti}, $\nabla \psi_{ti}$, and $\nabla^2 \psi_{ti}$ are treated as independent functions of $\mathbf{x}$ and $\mathbf{p}$ during this differentiation.

To compute the derivatives of the trial function ψ_{ti}, we must incorporate knowledge of the structure of the neural network. Given the trial function structure shown above, the general forms of these derivatives are:

$$\frac{\partial \psi_{ti}}{\partial p} = \frac{\partial}{\partial p} \left(A_i + P_i N_i \right) \tag{31}$$

$$= \frac{\partial A_i}{\partial p} + P_i \frac{\partial N_i}{\partial p} + \frac{\partial P_i}{\partial p} N_i \tag{32}$$

Noting that A_i and P_i are independent of the network parameters p:

$$\frac{\partial \psi_{ti}}{\partial p} = P_i \frac{\partial N_i}{\partial p} \tag{33}$$

For the gradient components, we have:

$$\nabla_{ij} \psi_{ti} = \frac{\partial \psi_{ti}}{\partial x_{ij}} = \frac{\partial}{\partial x_{ij}} (A_i + P_i N_i) = \frac{\partial A_i}{\partial x_{ij}} + P_i \frac{\partial N_i}{\partial x_{ij}} + \frac{\partial P_i}{\partial x_{ij}} N_i \tag{34}$$

$$\begin{aligned}\frac{\partial \nabla_{ij} \psi_{ti}}{\partial p} &= \frac{\partial^2 \psi_{ti}}{\partial p \partial x_{ij}} \\ &= \frac{\partial}{\partial p} \left(\frac{\partial A_i}{\partial x_{ij}} + P_i \frac{\partial N_i}{\partial x_{ij}} + \frac{\partial P_i}{\partial x_{ij}} N_i \right) \\ &= P_i \frac{\partial^2 N_i}{\partial p \partial x_{ij}} + \frac{\partial P_i}{\partial x_{ij}} \frac{\partial N_i}{\partial p}\end{aligned} \tag{35}$$

For the Laplacian components, we have:

$$\begin{aligned}\nabla^2_{ij}\psi_{ti} &= \frac{\partial^2\psi_{ti}}{\partial x_{ij}^2} \\ &= \frac{\partial}{\partial x_{ij}}\left(\frac{\partial A_i}{\partial x_{ij}} + P_i\frac{\partial N_i}{\partial x_{ij}} + \frac{\partial P_i}{\partial x_{ij}}N_i\right) \\ &= \frac{\partial^2 A_i}{\partial x_{ij}^2} + P_i\frac{\partial^2 N_i}{\partial x_{ij}^2} + 2\frac{\partial P_i}{\partial x_{ij}}\frac{\partial N_i}{\partial x_{ij}} + \frac{\partial^2 P_i}{\partial x_{ij}^2}N_i\end{aligned} \tag{36}$$

$$\begin{aligned}\frac{\partial\nabla^2_{ij}\psi_{ti}}{\partial p} &= \frac{\partial^3\psi_{ti}}{\partial p\partial^2 x_{ij}} \\ &= \frac{\partial}{\partial p}\left(\frac{\partial^2 A_i}{\partial x_{ij}^2} + P_i\frac{\partial^2 N_i}{\partial x_{ij}^2} + 2\frac{\partial P_i}{\partial x_{ij}}\frac{\partial N_i}{\partial x_{ij}} + \frac{\partial^2 P_i}{\partial x_{ij}^2}N_i\right) \\ &= P_i\frac{\partial^3 N_i}{\partial p\partial x_{ij}^2} + 2\frac{\partial P_i}{\partial x_{ij}}\frac{\partial^2 N_i}{\partial p\partial x_{ij}} + \frac{\partial^2 P_i}{\partial x_{ij}^2}\frac{\partial N_i}{\partial p}\end{aligned} \tag{37}$$

At this point, we must define the analytical forms of the functions $A(\mathbf{x})$, $P(\mathbf{x})$, and $N(\mathbf{x}, \mathbf{p})$ to complete the calculation of the error gradients.

2.3 Boundary condition function $A(\mathbf{x})$

Previous work with this technique has required development of custom forms for $P(\mathbf{x})$ and $A(\mathbf{x})$. The difficulty of finding boundary condition functions $A(\mathbf{x})$ which satisfy the above restrictions greatly increases with problem dimensionality. In the proposed work, we offer a new analytical technique for automatically generating the boundary condition function, which allows the solution of differential equations using neural networks to be automated and generalized.

The boundary condition function must satisfy the boundary conditions when evaluated at the boundaries, but otherwise may have an arbitrary value within the problem domain. The boundary conditions, and thus the boundary condition function, must also be continuous throughout the boundary surface. Here we develop the general procedure for automating the construction of the boundary condition function for a problem of arbitrary dimensionality with Dirichlet boundary conditions (future work will incorporate Neumann and other boundary condition types). The general approach is recursive, with the form for m dimensions incorporating the form for $m-1$ dimensions.

2.3.1 One-Dimensional Boundary Value Problem

For the one-dimensional boundary value problem, the form of $A(x)$ can be written as a sum of one term per boundary (assuming a unit domain; modification for non-unit domains is straightforward):

$$A(x) = A_1(x) = F_0(x) + F_1(x) \tag{38}$$

where the (otherwise arbitrary) functions $F_0(x)$ and $F_1(x)$ must allow $A_1(x)$ to satisfy:

1. $A_1(0) = f_0(0) = \text{constant}$
2. $A_1(1) = f_1(1) = \text{constant}$

where $f_0(0)$ and $f_1(1)$ are Dirichlet boundary conditions at $x = 0$ and $x = 1$. The simplest general form which satisfies these conditions is:

$$\begin{aligned} F_0(x) &= (1-x)f_0(0) \\ F_1(x) &= xf_1(1) \\ A_1(x) &= (1-x)f_0(0) + xf_1(1) \end{aligned} \tag{39}$$

As desired, this expression for $A_1(x)$ results in $f_0(0)$ at $x = 0$ and $f_1(1)$ at $x = 1$.

2.3.2 Two-Dimensional Boundary Value Problem

The two-dimensional analog of this equation is constructed using a similar approach, and utilizes the requirement of boundary condition continuity at axis intersections. Consider a two-dimensional PDE of variables x and y with Dirichlet boundary conditions:

$$\begin{aligned} \psi(0,y) &= f_0(0,y) \\ \psi(1,y) &= f_1(1,y) \\ \psi(x,0) &= g_0(x,0) \\ \psi(x,1) &= g_1(x,1) \end{aligned} \tag{40}$$

Continuity of boundary conditions requires that:

$$\begin{aligned} f_0(0,0) &= g_0(0,0) \\ f_1(1,0) &= g_0(1,0) \\ f_1(1,1) &= g_1(1,1) \\ f_0(0,1) &= g_1(0,1) \end{aligned} \tag{41}$$

If we begin with the 1-D form $A_1(x)$, and add a second variable y, it must remain valid at a fixed value of y, since y then acts simply as a constant in the equation:

$$A_1(x,y) = (1-x)f_0(0,y) + xf_1(1,y) \tag{42}$$

This equation retains the property of resulting in the boundary conditions at the x boundaries. Using this form as a starting point, we now expand the domain

of A from $0 \leq x \leq 1$ to $0 \leq x, y \leq 1$) using the same construction approach as for the one-dimensional case:

$$\begin{aligned} A_2(x,y) &= A_1(x,y) + G_0(x,y) + G_1(x,y) \\ &= (1-x)f_0(0,y) + xf_1(1,y) + G_0(x,y) + G_1(x,y) \end{aligned} \tag{43}$$

The relation for $A_2(x,y)$ must satisfy the boundary conditions (40):

$$\begin{aligned} A_2(0,y) &= f_0(0,y) \\ A_2(1,y) &= f_1(1,y) \\ A_2(x,0) &= g_0(x,0) \\ A_2(x,1) &= g_1(x,1) \end{aligned} \tag{44}$$

We can rewrite the arbitrary functions $G_0(x,y)$ and $G_1(x,y)$ to use a form analogous to the 1-D form, incorporating the second set of boundary conditions $g_0(x,0)$ and $g_1(x,1)$:

$$A_2(x,y) = A_1(x,y) + (1-y)(g_0(x,0) - G_0(x,y)) + y(g_1(x,1) - G_1(x,y)) \tag{45}$$

The additional terms for the y-dimension are required so that the x-dimension terms may be canceled out at the y-boundaries. At this point, the function $A_2(x,y)$ does not necessarily evaluate to the boundary conditions at the boundaries, since $G_0(x,y)$ and $G_1(x,y)$ are undefined. To allow $A_2(x,y)$ to satisfy the boundary conditions (40), the following must hold:

$$\begin{aligned} x = 0 : A_2(0,y) &= f_0(0,y) \\ &= f_0(0,y) + (1-y)(g_0(0,0) - G_0(0,y)) + y(g_1(0,1) - G_1(0,y)) \\ x = 1 : A_2(1,y) &= f_1(1,y) \\ &= f_1(1,y) + (1-y)(g_0(1,0) - G_0(1,y)) + y(g_1(1,1) - G_1(1,y)) \\ y = 0 : A_2(x,0) &= g_0(x,0) \\ &= (1-x)f_0(0,0) + xf_1(1,0) + g_0(x,0) - G_0(x,0) \\ y = 1 : A_2(x,1) &= g_1(x,1) \\ &= (1-x)f_0(0,1) + xf_1(1,1) + g_1(x,1) - G_1(x,1) \end{aligned} \tag{46}$$

Canceling the common terms on each side, we get:

$$\begin{aligned} x = 0 &: (1-y)(g_0(0,0) - G_0(0,y)) + y(g_1(0,1) - G_1(0,y)) = 0 \\ x = 1 &: (1-y)(g_0(1,0) - G_0(1,y)) + y(g_1(1,1) - G_1(1,y)) = 0 \\ y = 0 &: (1-x)f_0(0,0) + xf_1(1,0) - G_0(x,0) = 0 \\ y = 1 &: (1-x)f_0(0,1) + xf_1(1,1) - G_1(x,1) = 0 \end{aligned} \tag{47}$$

These equations are satisfied by the solutions:

$$\begin{aligned} G_0(x,y) &= (1-x)f_0(0,0) + xf_1(1,0)) = A_1(x,0) \\ G_1(x,y) &= (1-x)f_0(0,1) + xf_1(1,1)) = A_1(x,1) \end{aligned} \tag{48}$$

The 2-D formula now becomes:

$$A_2(x,y) = A_1(x,y) + (1-y)(g_0(x,0) - A_1(x,0)) + y(g_1(x,1) - A_1(x,1)) \tag{49}$$

Expanding the terms provides:

$$\begin{aligned} A_2(x,y) =& (1-x)f_0(0,y) + xf_1(1,y) + \\ & (1-y)(g_0(x,0) - ((1-x)f_0(0,0) + xf_1(1,0))) + \\ & y(g_1(x,1) - ((1-x)f_0(0,1) + xf_1(1,1))) \end{aligned} \tag{50}$$

and applying the boundary condition continuity relations, this becomes:

$$\begin{aligned} A_2(x,y) =& (1-x)f_0(0,y) + xf_1(1,y) + \\ & (1-y)(g_0(x,0) - ((1-x)g_0(0,0) + xg_0(1,0))) + \\ & y(g_1(x,1) - ((1-x)g_1(0,1) + xg_1(1,1))) \end{aligned} \tag{51}$$

This relation satisfies the requirements of the boundary condition function: it evaluates to the boundary conditions at the appropriate boundaries.

2.3.3 Three (and more) dimensions

The same procedure can be used to construct the boundary condition function for higher-dimensional spaces. For the three-dimensional case, with $h_0(x,y,0)$ and $h_1(x,y,1)$ being the boundary conditions for $z = 0, 1$:

$$A_3(x,y,z) = A_2(x,y,z) + (1-z)(h_0(x,y,0) - A_2(x,y,0)) + z(h_1(x,y,1) - A_2(x,y,1)) \tag{52}$$

Written in general form, for an m-dimensional space, the boundary condition function $A_m(\mathbf{x})$ is built from all $A_j(\mathbf{x})$ $(1 \leq j < m)$, and all A_j are functions of all m coordinates:

$$\begin{aligned} A_1(\mathbf{x}) =& (1-x_1)f_0(\mathbf{x}, x_1 = 0) + x_1 f_1(\mathbf{x}, x_1 = 1) \\ A_j(\mathbf{x}) =& A_{j-1}(\mathbf{x}) + \\ & (1-x_j)(B_{j0}(\mathbf{x}, x_j = 0) - A_{j-1}(\mathbf{x}, x_j = 0)) + \\ & x_j(B_{j1}(\mathbf{x}, x_j = 1) - A_{j-1}(\mathbf{x}, x_j = 1)) \end{aligned} \tag{53}$$

where B_{j0} and B_{j1} are the boundary conditions for variable j at $x_m = 0$ and $x_m = 1$, respectively. The expression (53) can be evaluated for arbitrary dimensionality m. Note that in all cases, if a boundary condition is not specified at a particular boundary (such as an initial value problem), the term for that boundary is dropped from the expression for the boundary condition function.

2.4 Network coefficient function $P(\mathbf{x})$

The simplest form of $P(\mathbf{x})$ that vanishes at the boundaries $x_j = 0|1 \ \forall \ j$ is:

$$P(\mathbf{x}) = \prod_{j=1}^{m} x_j(1 - x_j) \tag{54}$$

If no boundary condition is specified at a particular boundary, the term for that boundary is omitted from the product. Equation (54) ensures that the network output $N(\mathbf{x}, \mathbf{p})$ makes no contribution to the trial solution at the domain boundaries.

2.5 Network Output N

An example of the neural network structure used in the current work is illustrated in Figure 2.5. In a feedforward network of this type, the input signals $\mathbf{x}$ are sent to each hidden node via distinct connections, where they are weighted, summed, and transformed as discussed previously. The outputs from the hidden layer are then combined at the single output node in the same fashion, but without a bias, and an identity transfer function:

$$N = \sum_{k=1}^{H} v_k \sigma_k \tag{55}$$

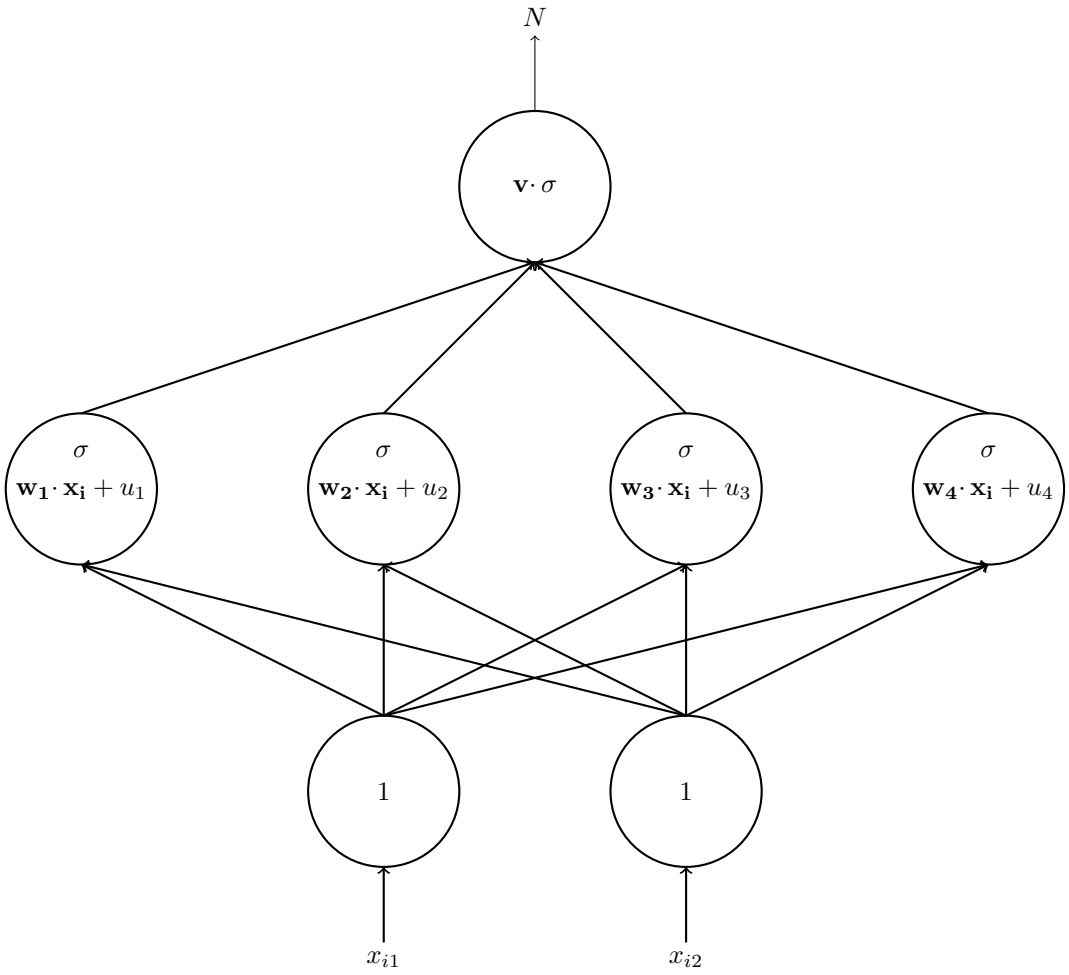

Figure 4: Schematic of a simple feed-forward neural network.

3 Methods and Techniques

3.1 The nnode software

The general approach described in the previous sections has been implemented in a library of Python code called `nnode`. It has been successfully applied to a variety of simple ordinary and partial differential equations. Below are examples of use of this code to solve one- and two-dimensional diffusion problems.

Note that these problems are initial value problems in time. In such cases, the second term for the time variable is dropped in the construction of the boundary condition function.

3.2 Examples

3.2.1 The Diffusion Equation

Consider the diffusion equation (Equation XXX):

$$G\left(\mathbf{x}, \psi, \nabla\psi, \nabla^2\psi\right) = \frac{\partial\psi}{\partial t} - D\nabla^2\psi = 0 \tag{56}$$

3.2.2 The One-Dimensional Diffusion Equation

In one spatial dimension, this becomes (note that $\mathbf{x} = (x, t)$:

$$G\left(\mathbf{x}, \psi, \nabla\psi, \nabla^2\psi\right) = \frac{\partial\psi}{\partial t} - D\frac{\partial^2\psi}{\partial^2 x} = 0 \tag{57}$$

Assume the presence of Dirichlet boundary conditions:

$$\begin{aligned} \psi(0,t) &= 0 \\ \psi(1,t) &= 0 \\ \psi(x,0) &= \sin(\pi x) \end{aligned} \tag{58}$$

This problem has an analytical solution (not in general true for PDEs):

$$\psi_a(\mathbf{x}) = e^{-\pi^2 Dt}\sin(\pi x) \tag{59}$$

With these boundary conditions, the boundary condition function, network coefficient function, and trial function reduce to:

$$\begin{aligned} A(\mathbf{x}) &= (1-t)\sin(\pi x) \\ P(\mathbf{x}) &= x(1-x)t \\ \psi_t(\mathbf{x}, \mathbf{p}) &= (1-t)\sin(\pi x) + x(1-x)tN(\mathbf{x}, \mathbf{p}) \end{aligned} \tag{60}$$

A neural network was constructed using $H = 10$ hidden nodes. All weights and biases were initialized with uniform random numbers in the range $[-1, 1]$. A uniform $10x10$ grid of training points in the domain $0 \leq x, t \leq 1$ was created for training.

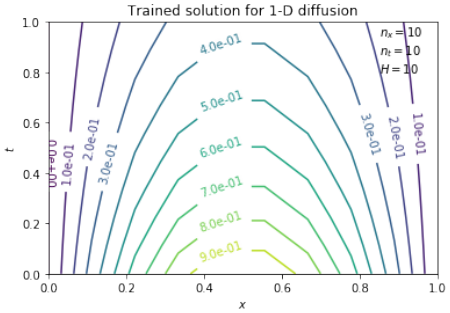

Figure 5: Trained solution for one-dimensional diffusion problem.

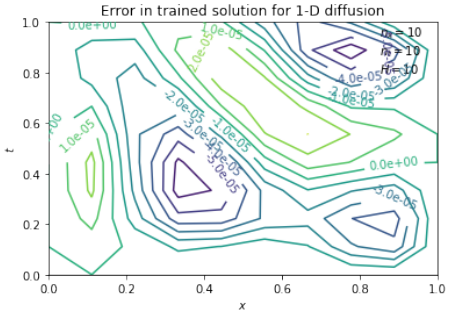

Figure 6: Error in trained solution for one-dimensional diffusion problem.

3.2.3 The Two-Dimensional Diffusion Equation

In this case, the diffusion equation may be written as:

$$G\left(\mathbf{x}, \psi, \nabla\psi, \nabla^2\psi\right) = \frac{\partial\psi}{\partial t} - D\left(\frac{\partial^2\psi}{\partial^2 x} + \frac{\partial^2\psi}{\partial^2 y}\right) = 0 \tag{61}$$

Assume the presence of the following Dirichlet boundary conditions:

$$\begin{aligned} \psi(0, y, t) &= 0 \\ \psi(1, y, t) &= 0 \\ \psi(x, 0, t) &= 0 \\ \psi(x, 1, t) &= 0 \\ \psi(x, y, 0) &= \sin(\pi x)\sin(\pi y) \end{aligned} \tag{62}$$

This problem has an analytical solution (not in general true for PDEs):

$$\psi_a(\mathbf{x}) = e^{-2\pi^2 Dt}\sin(\pi x)\sin(\pi y) \tag{63}$$

With these boundary conditions, the boundary condition function, network coefficient function, and trial function reduce to:

$$\begin{aligned} A(\mathbf{x}) &= (1-t)\sin(\pi x)\sin(\pi y) \\ P(\mathbf{x}) &= x(1-x)y(1-y)t \\ \psi_t(\mathbf{x}, \mathbf{p}) &= (1-t)\sin(\pi x)\sin(\pi y) + x(1-x)y(1-y)tN(\mathbf{x}, \mathbf{p}) \end{aligned} \tag{64}$$

A neural network was constructed using $H = 10$ hidden nodes. All weights and biases were initialized with uniform random numbers in the range $[-1, 1]$. A uniform $10x10x10$ grid of training points in the domain $0 \leq x, y, t \leq 1$ was created for training.

3.2.4 The One-Dimensional Diffusion Equation With Time-Varying Boundary Conditions

This problem is similar to the previous one-dimensional problem, but now the Dirichlet boundary condition at $x = 1$ is a function of time:

$$\begin{aligned} \psi(0, t) &= 0 \\ \psi(1, t) &= 0.1t \\ \psi(x, 0) &= \sin(\pi x) \end{aligned} \tag{65}$$

The same network architecture and initialization procedure was used as that for the original one-dimensional diffusion example.

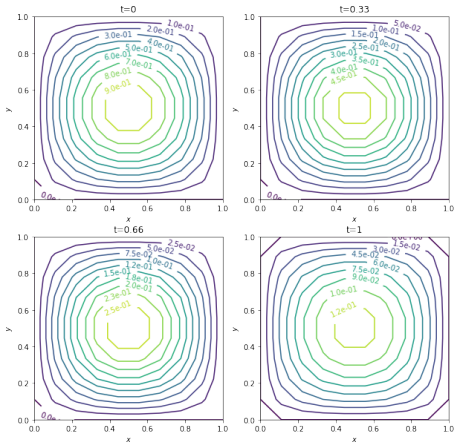

Figure 7: Trained solution for two-dimensional diffusion problem.

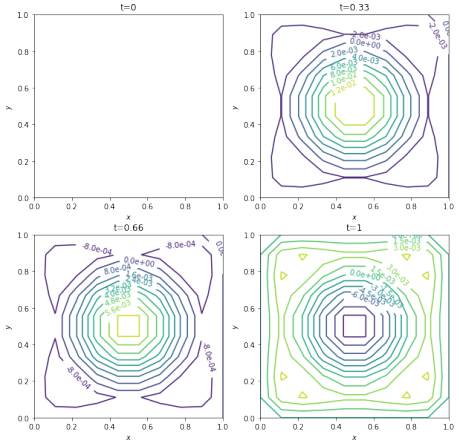

Figure 8: Error in trained solution for two-dimensional diffusion problem.

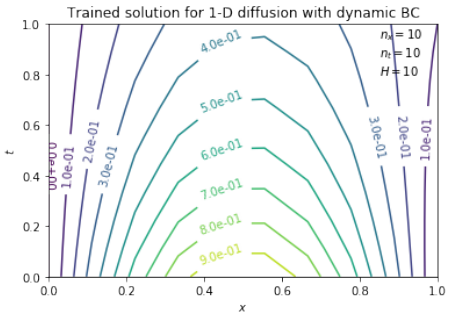

Figure 9: Trained solution for one-dimensional diffusion problem with dynamic BC.

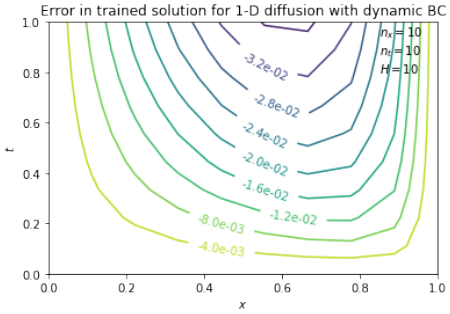

Figure 10: Error in trained solution for one-dimensional diffusion problem with dynamic BC.

3.3 Discussion

The examples shown above, and many others not shown, indicate that the initial version of the `nnode` software is capable of solving relatively simple partial differential equations with static and dynamic boundary conditions. The next section will describe the plans for extending this work to allow solution of the MHD equations.

4 Timeline

The existing `nnode` code will be enhanced in stages. At each stage, code will be tested using a standardized set of generic example problems, as well as the results of standard space physics models (see discussion in previous sections). These problems will include those with analytical solutions as well as those which require FDM or FEM solutions. Validation of results will be completely automated, and built into the test suite for the entire `nnode` package. Numerical solutions will be generated by 3rd-party data providers, e.g. CCMC, or computed with a standard PDE solution package, such as the NDSolve() function in Mathematica.

4.1 Select Reference Scenarios

Planned completion date: April 2020

In this stage, a definitive list of reference problems, both generic and space weather-specific, will be documented. Generic problems from the literature, e.g. [31] will be used, as well as a carefully-selected set of models from the space physics community, such as the CCMC and the SWPC.

4.2 Adapt Code to Multiprocessing System

Planned completion date: August 2020

In this stage, the code will be upgraded to take advantage of parallel computing facilities. Initially, this work will target the science computing cluster at the Space Telescope Science Institute in Baltimore, Maryland. This first stage will involve coarse parallelization - splitting the computation among available cores on the host machines. If time permits, the code will be further upgraded to take advantage of available accelerator technology, e.g. general-purpose graphical processing (GPGPU) cards. The progress of this latter capability will be dependent upon resource availability. If needed, existing GPGPU facilities at GMU will be utilized. Completion of this step will greatly broaden the scope of problems which can be addressed by the `nnode` software. Based on past experience with parallel codes, execution speed is expected to increase by at least an order of magnitude. The core of the code may undergo a redesign at this stage, to incorporate higher-performance machine learning libraries, such as TensorFlow and PyTorch.

4.3 Add Capability to Solve Systems of PDEs

Planned completion date: December 2020

This will be a relatively straightforward addition to the software. The modifications will start with simple aggregation of network outputs and errors in an overarching error function, and the consequent additional level of derivatives. This approach has already been shown to work in the literature [31].

4.4 Demonstrate Applicability to MHD Equations for Space Weather

Planned completion date: August 2021

A significant amount of time has been set aside for this phase of the work, since it constitutes the physical core of the investigation. During the prior development phases, a variety of well-understood space weather phenomena will be examined. One or more representative problems will be selected on the basis of simplicity, and the availability of real-world data against which the solution can be compared. These problems will then be used to demonstrate the real-world utility of the nnode software.

4.5 Write and Present Thesis

Planned completion date: December 2021

References

[1] Iftikhar Ahmad et al. "Intelligent computing to solve fifth-order boundary value problem arising in induction motor models". In: *Neural Computing and Applications* 29.7 (Aug. 2016), pp. 449–466. DOI: 10.1007/s00521-016-2547-6. URL: https://doi.org/10.1007/s00521-016-2547-6.

[2] Abir Alharbi et al. "An Artificial Neural Networks Method for Solving Partial Differential Equations". In: AIP, 2010. DOI: 10.1063/1.3498013. URL: https://doi.org/10.1063/1.3498013.

[3] William F. Allman. *Apprentices of Wonder: Inside the Neural Network Revolution*. Bantam, 1990. ISBN: 978-0553349467.

[4] Martin D. Altschuler and Gordon Newkirk. "Magnetic fields and the structure of the solar corona". In: *Solar Physics* 9.1 (Sept. 1969), pp. 131–149. DOI: 10.1007/bf00145734. URL: https://doi.org/10.1007/bf00145734.

[5] C.N. Arge and V.J. Pizzo. "Improvement in the prediction of solar wind conditions using near-real time solar magnetic field updates". In: *Journal of Geophysicsl Research* 105.A5 (May 2000), pp. 10465–10479.

[6] V.I. Avrutskiy. "Neural networks catching up with finite differences in solving partial differential equations in higher dimensions". In: *IEEE Transactions on Neural Networks and Learning Systems* VOLUME.NUMBER (2017), XXX–YYY.

[7] M. Baymani et al. "Artificial neural network method for solving the Navier–Stokes equations". In: *Neural Computing and Applications* 26.4 (Oct. 2014), pp. 765–773. DOI: 10.1007/s00521-014-1762-2. URL: https://doi.org/10.1007/s00521-014-1762-2.

[8] Jens Berg and Kaj Nyström. "A unified deep artificial neural network approach to partial differential equations in complex geometries". In: *Neurocomputing* 317 (Nov. 2018), pp. 28–41. DOI: 10.1016/j.neucom.2018.06.056. URL: https://doi.org/10.1016/j.neucom.2018.06.056.

[9] M.J. Berger and P. Colella. "Local Adaptive Mesh Refinement for Shock Hydrodynamics". In: *Journal of Computational Physics* 82 (1989), pp. 64–84.

[10] Edward K. Blum and Leong Kwan Li. "Approximation Theory and Feedforward Networks". In: *Neural Networks* 4 (1991), pp. 511–515.

[11] Pierre Cardaliaguet and Guillaume Euvrard. "Approximation of a Function and its Derivative with a Neural Network". In: *Neural Networks* 5 (1992), pp. 207–220.

[12] Brice Carnahan, H.A. Luther, and James O. Wilkes. *Applied Numerical Methods*. The University of Michigan, 1969. ISBN: 0471135070.

[13] NASA Community Coordinated Modeling Center. *CCMC: Community Coordinated Modeling Center*. URL: https://ccmc.gsfc.nasa.gov.

[14] Space Weather Prediction Center. *Homepage — NOAA/NWS Space Weather Prediction Center*. URL: https://swpc.noaa.gov.

[15] Space Weather Prediction Center. *WSA-ENLIL Solar Wind Prediction*. URL: https://swpc.noaa.gov/products/wsa-enlil-solar-wind-prediction.

[16] S. Chakraverty and Susmita Mall. *Artificial Neural Networks for Engineers and Scientists: Solving Ordinary Differential Equations*. CRC Press, 2017. ISBN: 1498781381.

[17] Francis F. Chen. *Introduction to Plasma Physics and Controlled Fusion 2e*. Plenum, 1984. ISBN: 0-306-41332-9.

[18] M. Chiaramonte and M. Kiener. "Solving differential equations using neural networks". In: Machine Learning Project, 2013.

[19] Daniela Danciu. "A CNN-based approach for a class of non-standard hyperbolic partial differential equations modeling distributed parameters (nonlinear) control systems". In: *Neurocomputing* 164 (Sept. 2015), pp. 56–70. DOI: 10.1016/j.neucom.2014.12.092. URL: https://doi.org/10.1016/j.neucom.2014.12.092.

[20] P.L. DeVries and J.E. Hasbun. *A First Course in Computational Physics*. Jones and Bartlett, 2011. ISBN: 978-0-7637-7314-4.

[21] Eirik Endeve. "2D MHD Models of the Large Scale Solar Corona". In: *AIP Conference Proceedings*. AIP, 2003. DOI: 10.1063/1.1618606. URL: https://doi.org/10.1063/1.1618606.

[22] GitHub. *GitHub*. URL: https://github.com.

[23] S. He, K. Reif, and R. Unbehauen. "Multilayer neural networks for solving a class of partial differential equations". In: *Neural Networks* 13 (2000), pp. 385–396.

[24] M.T. Heath. *Scientific Computing: An Introductory Survey 2e*. McGraw-Hill, 2002. ISBN: 0-07-239910-4.

[25] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2 (1989), pp. 359–366.

[26] K. Hornik, M. Stinchcombe, and H. White. "Universal Approximation of an Unknown Mapping and Its Derivatives Using Multilayer Feedforward". In: *Neural Networks* 3 (1990), pp. 551–560.

[27] Li Jianyu et al. "Numerical solution of elliptic partial differential equation using radial basis function neural networks". In: *Neural Networks* 16.5-6 (June 2003), pp. 729–734. DOI: 10.1016/s0893-6080(03)00083-2. URL: https://doi.org/10.1016/s0893-6080(03)00083-2.

[28] Neil R. Sheeley Jr. "Origin of the Wang–Sheeley–Arge solar wind model". In: *History of Geo- and Space Sciences* 8.1 (Mar. 2017), pp. 21–28. DOI: 10.5194/hgss-8-21-2017. URL: https://doi.org/10.5194/hgss-8-21-2017.

[29] Manoj Kumar and Neha Yadav. "Multilayer perceptrons and radial basis function neural network methods for the solution of differential equations: A survey". In: *Computers & Mathematics with Applications* 62.10 (Nov. 2011), pp. 3796–3811. DOI: 10.1016/j.camwa.2011.09.028. URL: https://doi.org/10.1016/j.camwa.2011.09.028.

[30] I.E. Lagaris, A. Likas, and D.I. Fotiadis. "Artificial Neural Network Methods in Quantum Mechanics". In: *Computer Physics Communications* 104 (1997), pp. 1–14.

[31] I.E. Lagaris, A. Likas, and D.I. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations". In: *IEEE Transactions on Neural Networks* 9.5 (1998), pp. 987–1000. DOI: 10.1109/72.712178. URL: https://doi.org/10.1109/72.712178.

[32] N. Mai-Duy and T. Tran-Cong. "Numerical solution of differential equations using multiquadric radial basis function networks". In: *Neural Networks* 14.NUMBER (2001), pp. 185–199. DOI: DOI. URL: URL.

[33] A. Malek and R. Shekari Beidokhti. "Numerical solution for high order differential equations using a hybrid neural network—Optimization method". In: *Applied Mathematics and Computation* 183.1 (Dec. 2006), pp. 260–271. DOI: 10.1016/j.amc.2006.05.068. URL: https://doi.org/10.1016/j.amc.2006.05.068.

[34] Susmita Mall and S. Chakraverty. "Numerical solution of nonlinear singular initial value problems of Emden–Fowler type using Chebyshev Neural Network method". In: *Neurocomputing* 149 (Feb. 2015), pp. 975–982. DOI: 10.1016/j.neucom.2014.07.036. URL: https://doi.org/10.1016/j.neucom.2014.07.036.

[35] Warren S. McCulloch and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 115–133.

[36] K.S. McFall and J.R. Mahan. "Artificial Neural Network Method for Solution of Boundary Value Problems With Exact Satisfaction of Arbitrary Boundary Conditions". In: *IEEE Transactions on Neural Networks* 20.8 (2009), pp. 1221–1233. DOI: DOI. URL: URL.

[37] A.J. Meade and A.A. Fernandez. "Solution of Nonlinear Ordinary Differential Equations by Feedforward Neural Networks". In: *Mathematical and Computer Modelling* 20.9 (1994), pp. 19–44.

[38] Siamak Mehrkanoon and Johan A.K. Suykens. "Learning solutions to partial differential equations using LS-SVM". In: *Neurocomputing* 159 (July 2015), pp. 105–116. DOI: 10.1016/j.neucom.2015.02.013. URL: https://doi.org/10.1016/j.neucom.2015.02.013.

[39] B. van Milligen, V. Tribaldos, and J.A. Jimenez. "Neural Network Differential Equation and Plasma Equilibrium Solver". In: *Physical Review Letters* 75.20 (1995), pp. 3594–3597. DOI: DOI. URL: URL.

[40] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969. ISBN: 9780262130431.

[41] Gianluca Di Muro and Silvia Ferrari. "A Constrained Backpropagation Approach to Solving Partial Differential Equations in Non-stationary Environments". In: IEEE, 2009, pp. 685–689.

[42] Dusan Odstrcil. *ENLIL: A Numerical Code for Solar Wind Disturbances*. URL: https://ccmc.gsfc.nasa.gov/models/Code_description.pdf.

[43] K. Rudd. "Solving Partial Differential Equations Using Artificial Neural Networks". In: *PhD thesis* (2013).

[44] Keith Rudd and Silvia Ferrari. "A constrained integration (CINT) approach to solving partial differential equations using artificial neural networks". In: *Neurocomputing* 155 (May 2015), pp. 277–285. DOI: 10.1016/j.neucom.2014.11.058. URL: https://doi.org/10.1016/j.neucom.2014.11.058.

[45] Keith Rudd, Gianluca Di Muro, and Silvia Ferrari. "A Constrained Backpropagation Approach for the Adaptive Solution of Partial Differential Equations". In: *IEEE Transactions on Neural Networks and Learning Systems* 25.3 (Mar. 2014), pp. 571–584. DOI: 10.1109/tnnls.2013.2277601. URL: https://doi.org/10.1109/tnnls.2013.2277601.

[46] C. T. Russell. *Space Physics: An Introduction*. Cambridge University Press, Aug. 2016. ISBN: 1107098823. URL: https://www.xarg.org/ref/a/1107098823/.

[47] Martin Schreiber et al. “Beyond spatial scalability limitations with a massively parallel method for linear oscillatory problems”. In: *The International Journal of High Performance Computing Applications* 32.6 (Feb. 2017), pp. 913–933. DOI: 10.1177/1094342016687625. URL: https://doi.org/10.1177/1094342016687625.

[48] Olena Schuessler and Diego Loyola. “Parallel Training of Artificial Neural Networks Using Multithreaded and Multicore CPUs”. In: *Adaptive and Natural Computing Algorithms*. Springer Berlin Heidelberg, 2011, pp. 70–79. DOI: 10.1007/978-3-642-20282-7_8. URL: https://doi.org/10.1007/978-3-642-20282-7_8.

[49] *Severe Space Weather Events–Understanding Societal and Economic Impacts*. National Academies Press, May 2009. DOI: 10.17226/12643. URL: https://doi.org/10.17226/12643.

[50] Nejib Smaoui and Suad Al-Enezi. “Modelling the dynamics of nonlinear partial differential equations using neural networks”. In: *Journal of Computational and Applied Mathematics* 170.1 (Sept. 2004), pp. 27–58. DOI: 10.1016/j.cam.2003.12.045. URL: https://doi.org/10.1016/j.cam.2003.12.045.

[51] The PETSc Team. *Portable, Extensible Toolkit for Scientific Computation*. URL: http:mcs.anl.gov/petsc/.

[52] A.R¿ Webb. “Functional Approximation by Feed- Forward Networks: A Least-Squares Approach to Generalization”. In: *IEEE Transactions on Neural Networksx* 5.3 (1994), pp. 363–371.

[53] WikiMedia. *Dendrite (PSF).svg*. URL: https://commons.wikimedia.org/wiki/File:Dendrite_(PSF).svg.

[54] WikiMedia. *Neural$_n$etwork.svg*. URL: https://commons.wikimedia.org/wiki/Neural_network#/media/File:Neural_network.svg.

[55] Eric Winter. *Neural network code for solving ordinary and partial differential equations*. URL: https://github.com/elwinter/nnode.

[56] Neha Yadav, Anupam Yadav, and Manoj Kumar. *An Introduction to Neural Network Methods for Differential Equations*. Springer Netherlands, 2015. DOI: 10.1007/978-94-017-9816-7. URL: https://doi.org/10.1007/978-94-017-9816-7.

[57] Hadi Sadoghi Yazdi, Hamed Modaghegh, and Morteza Pakdaman. “Ordinary differential equations solution in kernel space”. In: *Neural Computing and Applications* 21.S1 (May 2011), pp. 79–85. DOI: 10.1007/s00521-011-0621-7. URL: https://doi.org/10.1007/s00521-011-0621-7.

[58] Hadi Sadoghi Yazdi, Morteza Pakdaman, and Hamed Modaghegh. “Unsupervised kernel least mean square algorithm for solving ordinary differential equations”. In: *Neurocomputing* 74.12-13 (June 2011), pp. 2062–2071. DOI: 10.1016/j.neucom.2010.12.026. URL: https://doi.org/10.1016/j.neucom.2010.12.026.

[59] Yongqiang Zhou et al. “Global asymptotic stability analysis of nonlinear differential equations in hybrid bidirectional associative memory neural networks with distributed time-varying delays”. In: *Neurocomputing* 72.7-9 (Mar. 2009), pp. 1803–1807. DOI: 10.1016/j.neucom.2008.07.001. URL: https://doi.org/10.1016/j.neucom.2008.07.001.